Constant-Working-Space Algorithms (Computational Geometry and Discrete Mathematics)

AUTHOR(S):

Asano, Tetsuo

CITATION:

Asano, Tetsuo. Constant-Working-Space Algorithms (Computational Geometry and Discrete Mathematics). 数理解析研究所講究録 2009, 1641: 38-44

ISSUE DATE:

2009-04

URL:

http://hdl.handle.net/2433/140586

数理解析研究所講究録
第1641巻 2009年 38-44

Constant-Working-Space Algorithms

Tetsuo Asano
School of Information Science, JAIST, Japan

This talk presents a new direction of algorithms which do not use any extra working array. More formally, our goal is to design efficient algorithms which require no extra array of size depending on input size but use only constant working storage cells (variables), each having $\log n$ bits. This paper presents some new ideas for several problems related to binary image and computational geometry in the above-stated framework.

1 Introduction

Recent progress in computer systems has provided programmers with unlimited amount of working storage for their programs. Nowadays there are full of space-inefficient programs which use too much storage and becomes too slow if sufficiently large storage is not available. The author believes that there is high demand for space-efficient algorithms.

Another requirement of limited working storage comes from applications to built-in or embedded software in intelligent hardware. Digital cameras and scanners are good examples of intelligent hardware. We measure the space efficiency of an algorithm by the number of working storage cells (or the amount of working space) used by the algorithm. Ultimate efficiency is achieved when only constant number of variables are used in addition to input array(s).

We call such an algorithm *a constant-working-space algorithm*. Strictly speaking, there are two types of such algorithms. One should be rather referred to as an *in-place* algorithm. In this type of algorithms, input data are given by some constant number of arrays. Those arrays can be used as working space although there must be some upper limit on values to be stored in those arrays. Heap sort is a typical in-place algorithm. Ordinary implementation of mergesort requires a working array of the same size as the input array and thus it is not in-place. Quicksort does not require any array, but it is not in-place in the strict sense since its recursion depth depends on input size ($O(\log n)$ in average) which should be viewed as a part of the working space.

The other type of constant-working-space algorithms satisfy that condition in a more strict sense. That is, it should not use any working space of size depending on input size and an array storing input data is given as read-only memory so that any

value in the array cannot be changed. Constant-working-space algorithms for image processing in [1, 2, 3] are in-place algorithms in this sense. The algorithm for image scan with arbitrary angle [2] is a constant-working-space algorithm with input in read-only memory. The same framework has been studied in the complexity theory. A typical problem is a so-called *"st-connectivity"* problem: given an undirected graph G of n vertices in read-only memory and specified two arbitrary vertices s and t in G, determine whether they are connected or not using only constant number of variables of $O(\log n)$ bits. Reingold [7] succeeded in proving that the problem can be solved in polynomial time. It is a great break-through in this direction.

In this paper we briefly describe constant-working space algorithms for several problems on binary images, and some open problems concerning computational geometry.

2 Simple Example

Here is a simple example which explains a constant-working-space algorithm. Suppose we are given a linear array $a[\ ]$ with n numbers, $a[1], \ldots, a[n]$. Sum of consecutive elements in an interval $[i, j]$ with $1 \leq i \leq j \leq n$ is the sum of elements $a[i]$ through $a[j]$, that is, $\sum_{k=i}^{j} a[i]$. Given such an array, find the largest sum of consecutive elements. This is the problem addressed here.

Define another array by $s[i] = \sum_{k=1}^{i} a[k]$. Then, for any interval $[i, j]$ with $1 \leq i \leq j \leq n$, the corresponding sum is given by $s[j] - s[i-1]$ since $a[i] + \cdots + a[j] = (a[1] + \cdots + a[j]) - (a[i-1] + \cdots + a[j])$. Since our objective here is to find the largest sum, for each value of j we are only interested in an index i such that $[i]$ is smallest in the interval $[1, j-1]$, which is sometimes called the left-to-right minima. If we maintain the left-to-right minima for each index $j = 2, 3, \ldots, n$, we can find in constant time the index i in the interval $[1, j-1]$ that minimizes $s[i]$. Thus, we have a linear-time algorithm.

The linear algorithm uses a working array of size n. Is it possible to implement the same idea without using any extra array? The answer is yes. A key observation is that we do not need all the values in the array $a[\ ]$ but just one value that is smallest so far. In each iteration of the loop for $j = 2, \ldots, n$, we compute the sum $a[1] + \cdots + a[j]$ using the sum in the previous iteration and then if it is smaller than the previous minima then we update the minima. It is done in constant time and hence the running time remains linear. Note that we have used no extra array and also that we have never changed any value in the array. The input array was treated as a read-only array.

In this example, we had a linear-time algorithm under the constraint that the number of variables allowed is a constant, i.e., the size of the working space is a constant (or $O(\log n)$ bits) and an input array should be treated as a read-only array. As will be described later, the median-finding problem is also solved in linear time using $O(n)$ working space, but no linear-time algorithm is known if only constant working space is allowed. Is there any essential difference between the two problems? It is still open.

3 Known Results

In this section we introduce three major results in this framework.

Median Finding: Given a read-only array of n numbers, find their median using constant working space. A constant-working space algorithm is known for this problem which runs in $O(n^{1+\epsilon})$ time for any small constant $\epsilon > 0$ although it needs working space $O(1/\epsilon)$. This is a results by Munro and Raman in 1996 [6].

st-connectivity in graph: Given an undirected graph using a read-only array, determine whether two arbitrarily given vertices belong to the same connected component. Reingold [7] finally solved the long-standing open problem by giving a polynomial-time algorithm for this problem in 2005.

st-connectivity in binary image: Given a read-only array of a binary image, determine whether two arbitrarily specified pixels of the same value belong to the same connected component. This problem is quite similar to the st-connectivity in graph, but it is much simpler in the sense that the corresponding graph is 2-regular. A simple but efficient algorithm was given by Malgouyresa and Moreb [5] in 2002.

4 Median Finding

Median finding is one of the most fundamental problems in algorithms. It is well known that the median among n numbers can be computed in linear time [4]. In 1996, Munro and Raman [6] designed an almost linear-time algorithm using a constant amount of working space. More precisely, their algorithm runs in $O(n^{1+\epsilon})$ time for any small constant $\epsilon > 0$ using working space $O(1/\epsilon)$.

What happens if we can use $O(\sqrt{n})$ working space? We can design an $O(n \log n)$ time algorithm in this case. First, we partition an array into $\sqrt{n}$ blocks $B_1, B_2, \ldots, B_{\sqrt{n}}$ each contains $O(\sqrt{n})$ elements. In each block we apply the linear-time algorithm [4] to find a block median. We store $\sqrt{n}$ block medians in a working array and then find their median m_1 again using the same algorithm. Then, we compute the rank of m_1 by counting the number of elements smaller than m_1 while scanning the entire input array. If the rank is greater $n/2$ then all the elements greater than m_1 can be discarded from further search. Otherwise, we discard all the elements smaller than m_1. Since m_1 is the median of block medians, the number of discarded elements is at least $n/4$. We apply the same algorithm to the remaining elements. Then, again we can discard at least quarter of the remaining elements. Thus, after $\log n$ iteration we can locate the median. The running time of the algorithm is $O(n \log n)$ since each iteration is done in $O(n)$ time and the number of iteration is $O(\log n)$.

It is not known whether there is a linear-time algorithm for median finding using only constant working space.

5 Some Problems on Binary Images

We start with the st-connectivity problem in a binary image listed above. Figure 1 shows an example of a binary image with each pixel having a value 0 or 1. There is a natural definition of a connected components of pixels valued 1, i.e., two pixels of value 1 are said to be connected if there is any sequence of pixels of value 1 interconnecting the two pixels in such a way that any two consecutive pixels in the sequence are horizontally or vertically adjacent. A maximal subset of mutually connected pixels forms a connected components.

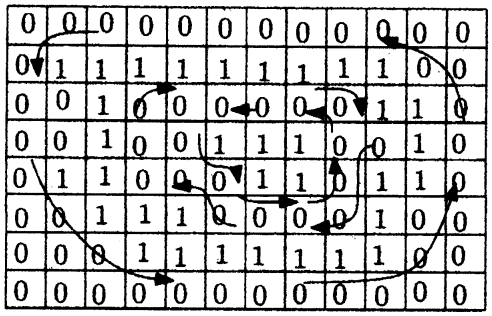

Figure 1: An example of a binary image containing two connected components of pixels valued 1. External boundaries are oriented in a counter-clockwise way while internal ones clockwisely ordered.

The first and most fundamental question is to ask whether two arbitrarily given pixels belong to the same connected component. It is rather easy to answer this question if sufficient working space is available. However, it is not trivial whether we can answer it in polynomial-time or not in the constant working space model with read-only arrays. Fortunately, Malgouyresa and Moreb [5] gave a polynomial-time algorithm. The idea is to define a canonical edge on each boundary. When we assume each boundary between 0 and 1 pixels is oriented so that '1' pixel always lies to its left, any external boundary of a connected component is oriented in a counter-clockwise order and any internal boundary is ordered clockwisely. Now, a **canonical edge** on a boundary is defined as an edge on the boundary which is lexicographically smallest in its y and x coordinates. With this convention, we are ready to describe the algorithm by Malgouyresa and Moreb [5].

Given a '1' pixel s, we keep walking horizontally to the left until we encounter a '0' pixel. Then, we follow the boundary from the edge until we return to the original place. Once we follow the boundary, we check whether it is external or internal. If it is external one, then we are done. Otherwise, we find its canonical edge and then again keep walking horizontally to the left until the first '0' pixel. After visiting some number of internal boundaries, we eventually arrive at an edge on an external boundary. Now, we associate the pixel s with the canonical edge of the external boundary. We repeat the same process for the other pixel t. The two pixels s and t belong to the same

component only if they are associated with the same canonical edge.

Note that we can follow a boundary following a local rule and also it is easily seen that constant number of variables are enough to determine the orientation of a boundary. Thus, the algorithm runs in $O(n)$ time for a binary image of n pixels.

Based on the idea behind the algorithm we can solve the following problems using only constant working space.

Connected Components Counting: Given a binary image, count the number of connected components of white pixels (of value 1) under 4- or 8-connectivity.

Computing Level of Component: A level of a connected component C_i is the number of connected components that enclose C_i. In a similar way we can define a level of a boundary B_j (internal or external) by the number of boundaries that enclose B_j.

Lowest Common Ancestor: Given two pixels of value 1, the problem is to find a connected component which is the lowest common ancestor in the corresponding inclusion tree.

6 Some Open Problems

A considerable amount of works have been done in the complexity theory under the name of *LogSpace*. It seems like that their interest is in deciding whether a given problem belong to *log space* or not, in other words, whether there is a polynomial-time algorithm for solving it using only working space of $O(\log n)$ bits. In this sense the median finding problem described earlier obviously belongs to *log space* since a naive algorithm runs in quadratic time. The author is interested in how fast we can implement such polynomial-time algorithms using constant working space with $O(\log n)$ bits in total. So, the median finding algorithm by Munro and Raman [6] is one typical example.

The following problems also belong to *log space*, but no subquadratic algorithm is known:

Min-gap: Given a read-only array of n numbers, find the minimum gap, which is the smallest difference between two consecutive numbers in their sorted order.

Max-gap: Given a read-only array of n numbers, find the maximum gap, which is the largest difference between two consecutive numbers in their sorted order.

Element Uniqueness Given a read-only array of n numbers, determine whether any two of them are equal or not.

Related geometric problems are as follows:

Diameter: Given a read-only array of n points in the plane, find a farthest pair of points.

Minimum Enclosing Circle: Given a read-only array of n points in the plane, find a circle of the smallest possible radius such that it encloses all given points in its interior and its center lies in the convex hull of those given points.

7 Conclusions and Future Works

In this paper we have proposed a new framework characterized by constant working space and read-only arrays. In the complexity theory this framework has been studied in a different name, i.e., log-space. Because of these constraints we have only polynomial-time algorithms. It is a main concern in the complexity theory whether a problem belongs to the class or not, in other words, whether there is any polynomial-time algorithm for a give problem or not. We are interested in how fast we can solve such a problem. This is one distinction.

8 Acknowledgement

Recently I have started to work on constant-working-space algorithms. This article refers to ongoing projects jointly with many researchers. Since there are too many of them, I have omitted their names from the authors. I would like to express my sincere gratitude to Sergey Bereg (Univ. of Texas at Dallas), Lilian Buzer (Univ. of Pairs, Est), Danny Chen (Notre Dame Univ.), Siu-Wing Cheng (Hong Kong UST), Otfried Cheong (KAIST), Alon Efrat (Univ. of Arizona), Xavier Goaoc (INRIA, Nancy), David Kirkpatrick (UBC), Masashi Kiyomi (JAIST), Jack Snoeyink (Univ. of North Carolina), and Hiroshi Tanaka (JAIST).

References

[1] T. Asano, S. Bitou, M. Motoki and N. Usui, "In-place algorithm for image rotation," Proc. ISAAC 2007, pp.704-715, Sendai, Dec. 2007.

[2] T. Asano, "Constant-working-space image scan with a given angle," Proc. 24th European Workshop on Computational Geometry, pp. 99-102, March 2008.

[3] T. Asano and H. Tanaka, "Constant-Working-Space Algorithm for Connected Component Labeling," Technical Report COMP-2008-01 of IEICE of Japan, 2008.

[4] M. Blum, R.W. Floyd, V. Pratt, R. Rivest and R. Tarjan, "Time bounds for selection," J. Comput. System Sci., 7, pp.448-461, 1973.

[5] R. Malgouyresa, M. Moreb, "On the computational complexity of reachability in 2D binary images and some basic problems of 2D digital topology," Theoretical Computer Science 283, pp.67-108, 2002.

[6] J. I. Munro and V. Raman, "Selection from read-only memory and sorting with minimum data movement," Theoretical Computer Science 165, pp.311-323, 1996.

[7] Omer Reingold, "Undirected st-connectivity in log-space," Proc. ACM Symp. on Theory of Computing, pp.376-385, 2005.